



下载APP



31 | 性能调优：手把手带你提升应用的执行性能

2021-05-24 吴磊

《Spark性能调优实战》

课程介绍 >



讲述：吴磊

时长 27:34 大小 25.25M



你好，我是吴磊。

在上一讲，我们一起完成了小汽车摇号趋势分析的应用开发，解决了 5 个案例。今天这一讲，我们逐一对这 5 个案例做性能调优，一起把专栏中学过的知识和技巧应用到实战中去。

由于趋势分析应用中的案例较多，为了方便对比每一个案例调优前后的性能效果，我们先来对齐并统一性能对比测试的方法论。



首先，我们的性能对比测试是以案例为粒度的，也就是常说的 Case By Case。然后，在每一个案例中，我们都有对比基准（Baseline）。**对比基准的含义是，在不采取任何调优方法的情况下，直接把代码交付执行得到的运行时间。**之后，对于每一个案例，我们会采取

一种或多种调优方法做性能优化，每一种调优方法都有与之对应的运行时间。最终，我们将不同调优方法的运行时间与对比基准做横向比较，来观察调优前后的性能差异，并分析性能提升 / 下降的背后原因。

话不多说，我们直接开始今天的课程吧！

运行环境

既然调优效果主要由执行时间来体现，那在开始调优之前，我们有必要先来交代一下性能测试采用的硬件资源和配置项设置。硬件资源如下表所示。

节点角色	CPU	内存	磁盘	网络
Master	2 cores	8 GB	30GB	10 Gbps
Worker	4 cores	16 GB	30GB	10 Gbps
Worker	4 cores	16 GB	30GB	10 Gbps



硬件资源配置

为了避免因为实验本身而等待太长的时间，我使用了比较强悍的机器资源。实际上，为了跑通应用，完成性能对比测试，你使用笔记本也可以。而且为了给后续调优留出足够空间，除了必需的运行资源设置以外，其他配置项全部保留了默认值，具体的资源配置如下表所示。

JVM进程	CPU	内存
Driver	--driver-cores 1	--driver-memory 5G
Executor	--executor-cores 3	--executor-memory 12G



另外，由于调优方法中涉及 AQE 和 DPP 这些 Spark 3.0 新特性，因此，我建议你使用 3.0 及以上的 Spark 版本来部署运行环境，我这边采用的版本号是 Spark 3.1.1。

接下来，我们就 Case By Case 地去回顾代码实现，分别分析 5 个案例的优化空间、可能的调优方法、方法的效果，以及它们与对比基准的性能差异。

案例 1 的性能调优：人数统计

首先，我们先来回顾案例 1。案例 1 的意图是统计摇号总人次、中签者人数，以及去掉倍率影响之后的摇号总人次，代码如下所示。

[复制代码](#)

```
1 val rootPath: String = _
2
3 // 申请者数据（因为倍率的原因，每一期，同一个人，可能有多个号码）
4 val hdfs_path_apply = s"${rootPath}/apply"
5 val applyNumbersDF = spark.read.parquet(hdfs_path_apply)
6 applyNumbersDF.count
7 // 中签者数据
8 val hdfs_path_lucky = s"${rootPath}/lucky"
9 val luckyDogsDF = spark.read.parquet(hdfs_path_lucky)
10 luckyDogsDF.count
11
12 // 申请者数据（去掉倍率的影响）
13 val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct
14 applyDistinctDF.count
```

从上面的代码实现中，我们不难发现，短短的几行代码共有 3 个 Actions，也就是 3 个不同数据集上的 count 操作，这 3 个 Actions 会触发 3 个 Spark Jobs。其中，前 2 个 Jobs 都是读取数据源之后立即计数，没什么优化空间。第 3 个 Job 是在 applyNumbersDF 之上做去重，然后再统计计数。结合上一讲对于不同案例的讲解，我们知道，applyNumbersDF、luckyDogsDF 和 applyDistinctDF 这 3 个数据集，在后续的案例中会被反复引用。

因为上述 3 个数据集的引用次数过于频繁，所以我们甚至都不用去计算“运行成本占比”，就可以判定：利用 Cache 一定有利于提升执行性能。

使用 Cache 的一般性原则：

如果 RDD/DataFrame/Dataset 在应用中的引用次数为 1，那么坚决不使用 Cache

如果引用次数大于 1，且运行成本占比超过 30%，应当考虑启用 Cache

因此，对于第 3 个 Job，我们可以利用 Cache 机制来提升执行性能。调优方法很简单，我们只需在 `applyNumbersDF.count` 之前添加一行代码：`applyNumbersDF.cache`。

由于这个案例中性能对比测试的关注点是第 3 个 Job，那为了方便横向对比，我们先把不相干的 Jobs 和代码去掉，整理之后的对比基准和调优代码如下表所示。

对比基准	调优代码
<pre>val rootPath: String = _ // 申请者数据（因为倍率的原因，每一期， // 同一个人，可能有多个号码） val hdfs_path_apply = s"\${rootPath}/apply" val applyNumbersDF = spark.read.parquet(hdfs_path_apply) // 申请者数据（去掉倍率的影响） val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct applyDistinctDF.count</pre>	<pre>val rootPath: String = _ // 申请者数据（因为倍率的原因，每一期，同一个人，可能有多个号码） val hdfs_path_apply = s"\${rootPath}/apply" val applyNumbersDF = spark.read.parquet(hdfs_path_apply) applyNumbersDF.cache applyNumbersDF.count // 申请者数据（去掉倍率的影响） val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct applyDistinctDF.count</pre>



对比基准与调优代码

然后，我们把这两份代码分别打包、部署和执行，并记录 `applyDistinctDF.count` 作业的执行时间，来完成性能对比测试，我把执行结果记录到了下面的表格中。

对比标的	执行作业	调优方法	运行时间
对比基准	applyDistinctDF.count	N/A	80230ms
调优代码	applyDistinctDF.count	利用Cache机制	66088ms



性能对比

从中我们可以看到，相较于对比基准，调优之后的执行性能提升了 20%。坦白地说，这样的提升是我们意料之中的。毕竟前者消耗的是磁盘 I/O，而调优之后计数作业直接从内存获取数据。

案例 2 的性能调优：摇号次数分布

接下来，我们再来分析案例 2。案例 2 分为两个场景，第一个场景用于统计申请者摇号批次数量的分布情况，第二个场景也是类似，不过它的出发点是中签者，主要用来解答“中签者通常需要摇多少次号才能中签”这类问题。

场景 1：参与摇号的申请者

我们先来回顾一下场景 1 的代码实现。仔细研读代码，我们不难发现，场景 1 是典型的单表 Shuffle，而且是两次 Shuffle。第一次 Shuffle 操作是以数据列“carNum”为基准做分组计数，第二次 Shuffle 是按照“x_axis”列再次做分组计数。

复制代码

```
1 val result02_01 = applyDistinctDF
2   .groupBy(col("carNum"))
3   .agg(count(lit(1)).alias("x_axis"))
4   .groupBy(col("x_axis"))
5   .agg(count(lit(1)).alias("y_axis"))
6   .orderBy("x_axis")
7
8 result02_01.write.format("csv").save("_")
```

因此，场景 1 的计算实际上就是 2 次 Word Count 而已，只不过第一次的 Word 是 “carNum”，而第二次的 Word 是 “x_axis”。那么，对于这样的 “Word Count”，我们都有哪些调优思路呢？

在配置项调优那一讲，我们专门介绍了 Shuffle 调优的一些常规方法，比如调整读写缓冲区大小、By Pass 排序操作等等。除此之外，我们的脑子里一定要有一根弦：**Shuffle 的本质是数据的重新分发，凡是有 Shuffle 操作的地方都需要关注数据分布。所以对于过小的数据分片，我们要有意识地对其进行合并。**再者，在案例 1 中我们提到，applyNumbersDF、luckyDogsDF 和 applyDistinctDF 在后续的案例中会被反复引用，**因此给 applyDistinctDF 加 Cache 也是一件顺理成章的事情。**

调优的思路这么多，那为了演示每一种调优方法的提升效果，我会从常规操作、数据分区合并、加 Cache 这 3 个方向出发，分别对场景 1 进行性能调优。不过，需要说明的是，咱们这么做的目的，一来是为了开阔调优思路，二来是为了复习之前学习过的调优技巧。

当然了，在实际工作中，我们一般没有时间和精力像现在这样，一个方法、一个方法去尝试。那么，效率最高的做法应该是遵循我们一直强调的调优方法论，也就是**先去应对木桶的短板、消除瓶颈，优先解决主要矛盾，然后在时间、精力允许的情况下，再去应对次短的木板。**

那么问题来了，你认为上述 3 种调优思路分别应对的是哪些“木板”？这些“木板”中哪一块是最短的？你又会优先采用哪种调优技巧？接下来，我们就带着这些问题，依次对场景 1 做调优。

思路 1：Shuffle 常规优化

刚刚咱们提到，Shuffle 的常规优化有两类：一类是 By Pass 排序操作，一类是调整读写缓冲区。而 By Pass 排序有两个前提条件：一是计算逻辑不涉及聚合或排序；二是 Reduce 阶段的并行度要小于参数 spark.shuffle.sort.bypassMergeThreshold 的设置值。显然，场景 1 不符合要求，计算逻辑既包含聚合也包含排序。所以，我们就只有调整读写缓冲区这一条路可走了。

实际上，读写缓冲区的调优也是有前提的，因为这部分内存消耗会占用 Execution Memory 内存区域，所以提高缓冲区大小的前提是 Execution Memory 内存比较充裕。

由于咱们使用的硬件资源比较强劲，而且小汽车摇号数据整体体量偏小，因此咱们还是有一些“资本”对读写缓冲区做调优的。具体来说，我们需要调整如下两个配置项：

spark.shuffle.file.buffer，Map 阶段写入缓冲区大小

spark.reducer.maxSizeInFlight，Reduce 阶段读缓冲区大小

由于读写缓冲区都是以 Task 为粒度进行设置的，因此调整这两个参数的时我们要小心一点，一般来说 50% 往往是个不错的开始，对比基准与优化设置如下表所示。

对比基准	优化设置
spark.shuffle.file.buffer = 32KB（默认值） spark.reducer.maxSizeInFlight = 48MB（默认值）	spark.shuffle.file.buffer = 48KB（32KB * 1.5） spark.reducer.maxSizeInFlight = 72MB（48MB * 1.5）



对比基准与优化设置

两组对比实验的运行时间，我记录到了下面的表格中。从中我们不难发现，上述两个参数的调整，对于作业端到端执行性能的影响不大。不过，这种参数调了半天，执行效率并没有显著提升的场景，肯定让你似曾相识。这个时候，最好的办法就是我们继续借助“木桶短板”“瓶颈”以及“调优方法论”，去尝试其他的调优思路。

对比标的	执行作业	调优方法	运行时间
对比基准	result02_01.write.format("csv").save("_")	N/A	110981ms
优化设置	result02_01.write.format("csv").save("_")	配置项调整	110027ms



性能对比

思路 2：数据分区合并

接着，我们再来说第二个思路，数据分区合并。首先，咱们先来一起分析一下，场景 1 到底存不存在数据分片过小的问题。为了方便分析，我们再来回顾一遍代码。因为场景 1 的

计算基于数据集 `applyDistinctDF`，所以要回答刚刚的问题，我们需要结合数据集 `applyDistinctDF` 的存储大小，以及 Shuffle 计算过后 Reduce 阶段的并行度一起来看。

[复制代码](#)

```
1 val result02_01 = applyDistinctDF
2   .groupBy(col("carNum"))
3   .agg(count(lit(1)).alias("x_axis"))
4
5   .groupBy(col("x_axis"))
6   .agg(count(lit(1)).alias("y_axis"))
7
8   .orderBy("x_axis")
9
10 result02_01.write.format("csv").save("_")
```

并行度由配置项 `spark.sql.shuffle.partitions` 决定，其默认大小为 200，也就是 200 个数据分区。而对于数据集存储大小的估算，我们需要用到下面的函数。

[复制代码](#)

```
1 def sizeNew(func: => DataFrame, spark: => SparkSession): String = {
2
3   val result = func
4
5   val lp = result.queryExecution.logical
6
7   val size = spark.sessionState.executePlan(lp).optimizedPlan.stats.sizeInByte
8
9   "Estimated size: " + size/1024 + "KB"
10
11 }
```

给定 `DataFrame`，`sizeNew` 函数可以返回该数据集在内存中的精确大小。把 `applyDistinctDF` 作为实参，调用 `sizeNew` 函数，返回的估算尺寸为 2.6 GB。将数据集尺寸除以并行度，我们就能得到 Reduce 阶段每个数据分片的存储大小，也就是 13 MB（也就是 2.6 GB / 200）。通常来说，数据分片的尺寸大小在 200 MB 左右为宜，13 MB 的分片尺寸显然过小。

在调度系统那一讲（第 5 讲），我们说过，如果需要处理的数据分片过小，相较于数据处理，Task 调度开销将变得异常显著，而这样会导致 CPU 利用降低，执行性能变差。因

此，为了提升 CPU 利用率进而提升整体的执行效率，我们需要对过小的数据分片做合并。这个时候，AQE 的自动分区合并特性就可以帮我们做这件事情。

不过，要想充分利用 AQE 的自动分区合并特性，我们还需要对相关的配置项进行调整。这里，你直接看场景 1 是怎么设置这些配置项的就可以了。

对比基准	优化设置
<pre>//AQE默认是禁用状态 spark.sql.adaptive.enabled = false （默认值）</pre>	<pre>//启动AQE spark.sql.adaptive.enabled = true //自动分区合并 spark.sql.adaptive.coalescePartitions.enabled = true （默认值） spark.sql.adaptive.advisoryPartitionSizeInBytes = 160MB/200MB/210MB/400MB spark.sql.adaptive.coalescePartitions.minPartitionNum = 6</pre>

 极客时间

对比基准与优化设置

一旦开启 AQE 机制，自动分区合并特性会自动生效。表格中的配置项有两个需要我们特别注意，一个是最小分区数 minPartitionNum，另一个是合并之后的目标尺寸 advisoryPartitionSizeInBytes。

我们先来看最小分区数，也就是 minPartitionNum。minPartitionNum 的含义，指的是分区合并之后的分区数量，不能低于这个参数设置的数值。由于我们计算集群配置的 Executors 个数为 6，为了保证每个 CPU 都不闲着、有活儿干，我们不妨把 minPartitionNum 也设置为 6。

接下来是分区合并的目标尺寸，我们刚刚说过，分区大小的经验值在 200 MB 左右，因此我们不妨把 advisoryPartitionSizeInBytes 设置为 200 MB。不过，为了对比不同分区大小对于执行性能的影响，我们可以多做几组实验。

配置项调整前后的几组实验效果对比如下，可以看到，调优后的运行时间有所缩短，这说明分区合并对于提升 CPU 利用率和作业的整体执行性能是有帮助的。仔细观察下表，我们至少有 3 点洞察。

对比标的	advisoryPartitionSizeInBytes	合并后的并行度	运行时间
对比基准	N/A	200（未合并）	110981ms
优化设置1	160 MB	18	95496ms
优化设置2	200 MB	13	90755ms
优化设置3	210 MB	12	82701ms
优化设置4	400 MB	6	136760ms



性能对比

- 并行度过高、数据分片过小，CPU 调度开销会变大，执行性能也变差。
- 分片粒度划分在 200 MB 左右时，执行性能往往是最优的。
- 并行度过低、数据分片过大，CPU 数据处理开销也会过大，执行性能会锐减。

思路 3：加 Cache

最后一个思路是加 Cache，这个调优技巧使用起来非常简单，我们在案例 1 已经做过演示，因此，这里直接给出优化代码和运行结果。

对比基准	调优代码
<pre>// 案例1部分代码：申请者数据（去掉倍率的影响） //val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct applyDistinctDF.count val result02_01 = applyDistinctDF .groupBy(col("carNum")) .agg(count(lit(1)).alias("x_axis")) .groupBy(col("x_axis")) .agg(count(lit(1)).alias("y_axis")) .orderBy("x_axis") result02_01.write.format("csv").save("_")</pre>	<pre>// 案例1部分代码：申请者数据（去掉倍率的影响） //val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct applyDistinctDF.cache applyDistinctDF.count val result02_01 = applyDistinctDF .groupBy(col("carNum")) .agg(count(lit(1)).alias("x_axis")) .groupBy(col("x_axis")) .agg(count(lit(1)).alias("y_axis")) .orderBy("x_axis") result02_01.write.format("csv").save("_")</pre>



对比基准与调优代码

可以看到，利用 Cache 机制做优化，作业执行性能提升得非常显著。

对比标的	执行作业	调优方法	运行时间
对比基准	result02_01.write.format("csv").save("_")	N/A	110981ms
优化设置	result02_01.write.format("csv").save("_")	利用Cache机制	38204ms



性能对比

到此为止，我们尝试了 3 种调优方法来对场景 1 做性能优化，分别是 Shuffle 读写缓冲区调整、数据分区合并，以及加 Cache。第 1 种方法针对的是，Shuffle 过程中磁盘与网络的请求次数；第 2 种方法的优化目标，是提升 Reduce 阶段的 CPU 利用率；第 3 种方法针对的是，数据集在磁盘中的重复扫描与重复计算。

实际上，根本不需要做定量分析，仅从定性我们就能看出，数据集的重复扫描与计算的开销最大。因此，在实际工作中，对于类似的“多选题”，我们自然要优先选择能够消除瓶颈的第 3 种方法。

场景 2：幸运的中签者

完成了场景 1 单表 Shuffle 的优化之后，接下来，我们再来看看场景 2，场景 2 的业务目标是获取中签者的摇号次数分布。我们先来回顾场景 2 的代码实现，场景 2 的计算涉及一次数据关联，两次分组、聚合，以及最终的排序操作。不难发现，除了关联计算外，其他计算步骤与场景 1 如出一辙。因此，对于场景 2 的优化，我们专注在第一步的数据关联，后续优化沿用场景 1 的调优方法即可。

复制代码

```
1 val result02_02 = applyDistinctDF
2   .join(luckyDogsDF.select("carNum"), Seq("carNum"), "inner")
3   .groupBy(col("carNum")).agg(count(lit(1)).alias("x_axis"))
4   .groupBy(col("x_axis")).agg(count(lit(1)).alias("y_axis"))
5   .orderBy("x_axis")
6
7 result02_02.write.format("csv").save("_")
```

参与关联的两张表分别是 applyDistinctDF 和 luckyDogsDF，其中 applyDistinctDF 是去重之后的摇号数据，luckyDogsDF 包含的是中签者的申请编号与批次号。

applyDistinctDF 包含 1.35 条数据记录，而 luckyDogsDF 仅仅包含 115 万条数据记录。很显然，二者之间的数据关联属于数仓中常见的“大表 Join 小表”。

遇到“大表 Join 小表”的计算场景，我们最先应该想到的调优技巧一定是广播变量。毕竟，我们一直都在不遗余力地强调 Broadcast Joins 的优势与收益。在这里，我再强调一次，你一定要掌握使用广播变量优化数据关联的调优技巧。毫不夸张地说，广播变量是“性价比”最高的调优技巧，且没有之一。

要利用广播变量来优化 applyDistinctDF 与 luckyDogsDF 的关联计算，我们需要做两件事情。**第一件，估算 luckyDogsDF 数据表在内存中的存储大小。第二件，设置广播阈值配置项 spark.sql.autoBroadcastJoinThreshold。**

对于分布式数据集的尺寸预估，我们还是使用 sizeNew 函数，把 luckyDogsDF 作为实参，调用 sizeNew 函数，返回的估算尺寸为 18.5MB。有了这个参考值，我们就可以设置广播阈值了。要把 applyDistinctDF 与 luckyDogsDF 的关联计算转化为 Broadcast Join，只要让广播阈值大于 18.5MB 就可以，我们不妨把这个参数设置为 20MB。

对比基准	优化设置
spark.sql.autoBroadcastJoinThreshold = 10MB（默认值）	spark.sql.autoBroadcastJoinThreshold = 20MB



对比基准与优化设置

我把配置项调整前后的实验结果记录到了如下表格，显然，相比默认的 Shuffle Sort Merge Join 实现机制，Broadcast Join 的执行性能更胜一筹。

对比标的	执行作业	调优方法	运行时间
对比基准	result02_02.write.format("csv").save("_")	N/A	113180ms
优化设置	result02_02.write.format("csv").save("_")	配置项调整	93155ms



案例 3 的性能调优：中签率的变化趋势

案例 3 的业务目标是洞察中签率的变化趋势，我们先来回顾代码。要计算中签率，我们需要分两步走。第一步，按照摇号批次，也就是 batchNum 分别对 applyDistinctDF 和 luckyDogsDF 分组，然后分别对分组内的申请者和中签者做统计计数。第二步，通过数据关联将两类计数做除法，最终得到每个批次的中签率。

[复制代码](#)

```
1 // 统计每批次申请者的人数
2 val apply_denominator = applyDistinctDF
3   .groupBy(col("batchNum"))
4   .agg(count(lit(1)).alias("denominator"))
5
6 // 统计每批次中签者的人数
7 val lucky_molecule = luckyDogsDF
8   .groupBy(col("batchNum"))
9   .agg(count(lit(1)).alias("molecule"))
10
11 val result03 = apply_denominator
12   .join(lucky_molecule, Seq("batchNum"), "inner")
13   .withColumn("ratio", round(col("molecule")/col("denominator"), 5))
14   .orderBy("batchNum")
15
16 result03.write.format("csv").save("_")
```

由于 2011 年到 2019 年总共有 72 个摇号批次，因此第一步计算得到结果集，也就是 apply_denominator 和 lucky_molecule 各自有 72 条数据记录。显然，两个如此之小的数据集做关联不存在什么调优空间。

因此，对于案例 3 来说，调优的关键在于第一步涉及的两个单表 Shuffle。关于单表 Shuffle 的调优思路与技巧，我们在案例 2 的场景 1 做过详细的分析与讲解，因此，applyDistinctDF 和 luckyDogsDF 两张表的 Shuffle 优化就留给你作为课后练习了。

案例 4 的性能调优：中签率局部洞察

与案例 3 不同，案例 4 只关注 2018 年的中签率变化趋势，我们先来回顾案例 4 的代码实现。

```
1 // 筛选出2018年的中签数据，并按照批次统计中签人数
2 val lucky_molecule_2018 = luckyDogsDF
3 .filter(col("batchNum").like("2018%"))
4 .groupBy(col("batchNum"))
5 .agg(count(lit(1)).alias("molecule"))
6
7 // 通过与筛选出的中签数据按照批次做关联，计算每期的中签率
8 val result04 = apply_denominator
9 .join(lucky_molecule_2018, Seq("batchNum"), "inner")
10 .withColumn("ratio", round(col("molecule")/col("denominator"), 5))
11 .orderBy("batchNum")
12
13 result04.write.format("csv").save("_")
```

从代码实现来看，案例 4 相比案例 3 唯一的改动，就是在 luckyDogsDF 做统计计数之前增加了摇号批次的过滤条件，也就是 `filter(col("batchNum").like("2018%"))`。你可能会说：“案例 4 的改动可以说是微乎其微，它的调优空间和调优方法应该和案例 3 没啥区别”。还真不是，还记得 Spark 3.0 推出的 DPP 新特性吗？添加在 luckyDogsDF 表上的这个不起眼的过滤谓词，恰恰让 DPP 有了用武之地。

在 DPP 那一讲，我们介绍过开启 DPP 的 3 个前提条件：

事实表必须是分区表，且分区字段（可以是多个）必须包含 Join Key

DPP 仅支持等值 Joins，不支持大于、小于这种不等值关联关系

维表过滤之后的数据集，必须要小于广播阈值，因此，你要注意调整配置项 `spark.sql.autoBroadcastJoinThreshold`

那么，这 3 个前提条件是怎么影响案例 4 的性能调优的呢？

首先，在上一讲，我们介绍过摇号数据的目录结构，`apply` 和 `lucky` 目录下的数据都按照 `batchNum` 列做了分区存储。因此，案例 4 中参与关联的数据表 `applyDistinctDF` 和 `luckyDogsDF` 都是分区表，且分区键 `batchNum` 刚好是二者做关联计算的 Join Key。其次，案例 4 中的关联计算显然是等值 Join。

最后，我们只要保证 `lucky_molecule_2018` 结果集小于广播阈值就可以触发 DPP 机制。2018 年只有 6 次摇号，也就是说，分组计数得到的 `lucky_molecule_2018` 只有 6 条数据记录，这么小的“数据集”完全可以放进广播变量。

如此一来，案例 4 满足了 DPP 所有的前提条件，利用 DPP 机制，我们就可以减少 applyDistinctDF 的数据扫描量，从而在整体上提升作业的执行性能。

对比基准	DPP优化
<pre>//val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct applyDistinctDF.cache applyDistinctDF.count</pre>	<pre>//val applyDistinctDF = applyNumbersDF.select("batchNum", "carNum").distinct applyDistinctDF.count</pre>

极客时间

对比基准与DPP优化

DPP 的核心作用在于**降低事实表 applyDistinctDF 的磁盘扫描量**，因此案例 4 的调优办法非常简单，只要把最初加在 applyDistinctDF 之上的 Cache 去掉即可，如上表右侧所示。同时，为了公平起见，对比基准不应该仅仅是让 DPP 失效的测试用例，而应该是 applyDistinctDF 加 Cache 的测试用例。与此同时，我们直接对比 DPP 的磁盘读取效率与 Cache 的内存读取效率，也能加深对 DPP 机制的认知与理解。

把上述两个测试用例交付执行，运行结果如下。可以看到，相较对比基准，在 DPP 机制的作用下，案例 4 端到端的执行性能有着将近 5 倍的提升。由此可见，数据集加 Cache 之后的内存读取，远不如 DPP 机制下的磁盘读取更高效。

对比标的	执行作业	调优方法	运行时间
对比基准	result04.write.format("csv").save("_")	添加Cache	70739ms
优化设置	result04.write.format("csv").save("_")	利用DPP机制	16504ms

极客时间

性能对比

案例 5 的性能调优：倍率分析

案例 5 也包含两个场景，场景 1 的业务目标是计算不同倍率下的中签人数，场景 2 与场景 1 相比稍有不同，它的目的是计算不同倍率下的中签比例。

尽管两个场景的计算逻辑有区别，但是调优思路与方法是一致的。因此，在案例 5 中，我们只需要对场景 1 的性能优化进行探讨、分析与对比，我们先来回顾一下场景 1 的代码实现。

 复制代码

```
1 val result05_01 = applyNumbersDF
2 .join(luckyDogsDF.filter(col("batchNum") >= "201601"))
3 .select("carNum", Seq("carNum", "inner"))
4 .groupBy(col("batchNum"), col("carNum"))
5 .agg(count(lit(1)).alias("multiplier"))
6 .groupBy("carNum")
7 .agg(max("multiplier").alias("multiplier"))
8 .groupBy("multiplier")
9 .agg(count(lit(1)).alias("cnt"))
10 .orderBy("multiplier")
11
12 result05_01.write.format("csv").save("_")
```

仔细研读代码之后，我们发现场景 1 的计算分为如下几个环节：

大表与小表的关联计算，且小表带过滤条件

按 batchNum 列做统计计数

按 carNum 列取最大值

按 multiplier 列做统计计数

在这 4 个环节当中，关联计算涉及的数据扫描量和数据处理量最大。因此，这一环节是案例 5 执行效率的关键所在。另外，除了关联计算环节，其他 3 个环节都属于单表 Shuffle 优化的范畴，这 3 个环节的优化可以参考案例 2 场景 1 的调优思路与技巧，咱们也不多说了。因此，对于案例 5 的性能优化，我们重点关注第一个环节，也就是 applyNumbersDF 与 luckyDogsDF 的关联计算。

仔细观察第一个环节的关联计算，我们发现关联条件中的 Join Key 是 carNum，而 carNum 并不是 applyNumbersDF 与 luckyDogsDF 两张表的分区键，因此，在这个关

联查询中，我们没有办法利用 DPP 机制去做优化。

不过，applyNumbersDF 与 luckyDogsDF 的内关联是典型的“大表 Join 小表”，对于这种场景，我们至少有两种方法可以将低效的 SMJ 转化为高效的 BHJ。

第一种办法是计算原始数据集 luckyDogsDF 的内存存储大小，确保其小于广播阈值，从而利用 Spark SQL 的静态优化机制将 SMJ 转化为 BHJ。第二种方法是确保过滤后的 luckyDogsDF 小于广播阈值，这样我们就可以利用 Spark SQL 的 AQE 机制来动态地将 SMJ 转化为 BHJ。

接下来，我们分别使用这两种方法来做优化，比较它们之间，以及它们与对比基准之间的性能差异。在案例 2 场景 2 中，我们计算过 luckyDogsDF 在内存中的存储大小是 18.5MB，因此，通过适当调节 spark.sql.autoBroadcastJoinThreshold，我们就可以灵活地在两种调优方法之间进行切换。

测试用例	用例介绍	参数设置
对比基准	默认的SMJ	spark.sql.autoBroadcastJoinThreshold = -1
调优方法1	Spark SQL静态转化BHJ	spark.sql.autoBroadcastJoinThreshold = 20MB
调优方法2	AQE动态转化BHJ	spark.sql.autoBroadcastJoinThreshold = 10MB（默认值）



对比基准与优化设置

将 3 种测试用例付诸执行，在执行效率方面，SMJ 毫无悬念是最差的，而 AQE 的动态优化介于 SMJ 与 Spark SQL 的静态转化之间。毕竟，AQE 的 Join 策略调整是一种“亡羊补牢、犹未为晚”的优化机制，在把 SMJ 调整为 BHJ 之前，参与 Join 的两张表的 Shuffle 计算已经执行过半。因此，它的执行效率一定比 Spark SQL 的静态优化更差。尽管如此，AQE 动态调整过后的 BHJ 还是比默认的 SMJ 要高效得多，而这也体现了 AQE 优化机制的价值所在。

对比标的	执行作业	调优方法	运行时间
对比基准	result05_01.write.format("csv").save("_")	N/A	130068ms
调优方法1	result05_01.write.format("csv").save("_")	配置项调整	31229ms
调优方法2	result05_01.write.format("csv").save("_")	配置项调整	64952ms



性能对比

小结

今天这一讲，我们结合以前学过的知识点与调优技巧，以小汽车摇号为例 Case By Case 地做性能优化。涉及的优化技巧有 Shuffle 读写缓冲区调整、加 Cache、预估数据集存储大小、Spark SQL 静态优化、AQE 动态优化（自动分区合并与 Join 策略调整）以及 DPP 机制。为了方便你对比，我把它们总结在了一张脑图里。不过，我更希望你能自己总结一下，这样才能记得更好。



最后我想说，很遗憾我们没能在这个实战里，把专栏中所有的调优技巧付诸实践，这主要是因为小汽车摇号应用相对比较简单，很难覆盖所有的计算与优化场景。对于那些未能付诸实战的调优技巧，只能靠你在平时的工作中去实践了。

不过，专栏的留言区和咱们的读者群，会一直为你敞开，尽管我不能做到立即回复，但我可以承诺的是，对于你的留言，我只会迟到、绝不缺席！

每日一练

- 1. 你能参考案例 2 场景 1，完成案例 3 中 applyDistinctDF 和 luckyDogsDF 两张表的单表 Shuffle 优化吗？
- 2. 你能参考案例 5 场景 1，综合运用 AQE、Broadcast Join 等调优技巧，对案例 5 场景 2 做性能优化吗？

期待在留言区看到你的优化结果，也欢迎你随时提问，我们下一讲见！

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励

👍 赞 2 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 应用开发：北京市小客车（汽油车）摇号趋势分析

下一篇 期末考试 | “Spark性能调优” 100分试卷等你来挑战！

更多学习推荐

云原生训练营

向技术要红利，4个月，挑战50万年薪

孟凡杰 eBay 资深架构师
Kubernetes 社区贡献者

立抢上新特惠



精选留言 (2)

💬 写留言



斯盖丸

2021-05-31

老师，我看到很多hdfs源的spark job，它们task建议的大小是128M，也就是一个HDFS block的大小。请问咱们这边说的一个task最佳大小是200多M是怎么来的？

展开

作者回复: 好问题，200M是无数次调优得来的经验值~ 128MB是HDFS的推荐大小，用在Spark的Task上，有些偏小了。



斯盖丸
2021-05-31

老师，请问数据分片200mb是最佳的，这个结论是如何得出的？因为我自己也试过几次发现确实如此，一旦分片数再大执行时间反而更慢。很好奇老师是怎么知道这个知识点的，我去stackoverflow上查过，没有什么收获

作者回复: 前面回复过了哈，无数次实战得到的经验值~